软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006069]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种适应 GPU 的混合访问缓存索引框架*

张鸿骏[1,2], 武延军[1], 张 珩[1], 张立波[1]

[1](中国科学院软件研究所,北京 100190)

[2](中国科学院大学,北京 100100)

通讯作者: 张鸿骏, E-mail: hongjun@iscas.ac.cn

摘 要: 散列表（Hash Table）作为一类根据关键码值(Key value)提供高效数据访问的数据索引结构,其广泛应用于各类计算机应用,尤其在对性能要求极高的系统软件、数据库以及高性能计算领域.在网络、云计算和物联网服务方面,以散列表为核心结构已经成为缓存系统的重要系统组件.然而,随着大规模数据量的大幅增加,以多核CPU为核心设计散列表结构的系统已经逐渐出现性能瓶颈,亟需进一步改进散列表的高性能和可扩展性.随着通用图形处理器（Graphic Processing Unit,GPU）的日益普及和硬件计算能力和并发性能大幅度提升,各类以并行计算为核心的系统软件任务在 GPU 上进行了优化设计并得到可观的性能提升.由于存在稀疏性和随机性,采用现有散列表的并行结构直接在 GPU 上应用势必会带来高频次的内存访问和频繁的总线数据传输,影响了散列表在 GPU 上的性能发挥.本文重点分析了缓存系统中散列表索引的内存访问、命中率与索引开销,提出并设计了一种适应GPU的混合访问缓存索引框架CCHT(Cache Cuckoo Hash Table),提供了两种适应不同命中率和索引开销要求的缓存策略,允许写入与查询操作并发执行,最大程度地利用了 GPU 硬件的计算性能与并发特性,减少了内存访问与总线传输.通过在 GPU 硬件上的实现与实验验证,CCHT 在保证缓存命中率的同时,性能优于其他用于缓存索引的散列表.

关键词: 系统软件,缓存索引,散列表,GPU

中图法分类号: TP311

Hybrid Access Cache Indexing Framework Adapted to the GPU

ZHANG Hong-Jun[1,2], WU Yan-Jun[1], ZHANG Heng[1], ZHANG Li-Bo[1]

[1](Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Hash tables, as a type of data indexing structure that provides efficient data access based on key values, are widely used in various computer applications, especially in system software, databases, and high performance that require high performance Computing field. In network, cloud computing and IoT services, hash tables have become the core system components of cache systems. However, with the large-scale increase in the amount of large-scale data, performance bottlenecks have gradually emerged in systems designed with a multi-core CPU as the core of the hash table structure. There is an urgent need to further improve the high performance and scalability of the hash table. With the increasing popularity of general-purpose graphics processing units (GPUs) and the substantial improvement of hardware computing capabilities and concurrency performance, various types of system software tasks with parallel computing as the core have been optimized on the GPU and have achieved considerable performance promotion. Due to the sparseness and randomness, using the existing parallel structure of the hash table directly on the GPU will inevitably bring high-frequency memory access and frequent bus

* 基金项目: 中国科学院战略性先导科技专项预研课题(Y8XD373105);中国科学院前沿科学重点研究计划(ZDBS-LY-JSC038)

Foundation item: the Strategic Priority Research Program of the Chinese Academy of Sciences (Y8XD373105); Key Research Program of Frontier Sciences, CAS (ZDBS-LY-JSC038)

收稿时间: 2020-02-10; 修改时间: 2020-04-04; 采用时间: 2020-05-09; jos 在线出版时间: 2020-06-10

data transmission, which affects the performance of the hash table on the GPU. This paper focuses on the analysis of memory access, hit rate, and index overhead of hash table indexes in the cache system. A hybrid access cache index framework CCHT (Cache Cuckoo Hash Table) adapted to GPU is proposed and provided. The cache strategy required by index and index overhead allows concurrent execution of write and query operations, maximizing the use of the computing performance and concurrency characteristics of GPU hardware, reducing memory access and bus transferring overhead. Through GPU hardware implementation and experimental verification, CCHT has better performance than other cache indexing hash table while ensuring cache hit rate.

Key words: system software; cache index; hash table; GPU

在系统软件与高性能数据应用中,散列表（Hash Table）是一类重要和常见的数据索引结构.通过把关键码值(Key value)映射到表中一个内容位置来访问对应的数据记录,以大幅度加快数据查找的速度.在内存关系型数据库[1-3]和关键码值类型数据库[4,5]领域,散列表作为核心的系统组件提供了关键的数据索引接结构,通过映射函数将数据索引至对应位置.具体地,网络、云计算和物联网服务的重要系统组件[32],基于键值的内存缓存系统,如 memcached[5],Ramcloud[8]和 MemC3[9],都采用了散列表来提供内存内的高性能数据索引服务.内存缓存系统通过将数据从相对内存访问速度慢的磁盘等存储介质加载入内存,以提升应用程序对数据访问的性能.由于内存通常不能容纳所有数据,当内存空间满时,缓存替换系统通过缓存替换算法踢出部分缓存数据为新数据提供空间.

在散列表的现有工作中,基于 cuckoo 算法的散列表 CHT（Cuckoo Hash Table）[6]通过将原有单一数据映射到多个桶（Bucket）中.CHT 代替了传统散列表的映射与链表式操作的方法,具有快速访问与节省空间的特征,在散列表管理[16,26,33]和数据存储系统[4,22-25,9]上得到了广泛的应用与研究.

典型的 CHT 不支持插入和查询操作同时发生.它将数据通过两个不同的散列算法映射到两个不同的桶中,每个桶中有四个槽（Slot）允许存放数据.每个数据对应一组键值数据（Key-Value Pair）.在执行插入操作时,判断映射桶中是否有空闲槽,如果有则进行插入;如果没有空闲槽,则通过对现有单元中存储数据进行散列与替换获得空闲槽进行插入.进而,如果在键值对数据插入时需要替换槽中数据,CHT 会随机选择一个待替换键值对 KV’进行置换,通过散列算法计算 KV’对应另一可选桶,将 KV 放入原有 KV’所在槽中;如果另一可选桶有空闲槽,则将 KV’放入该空间槽中;如果该桶没有空闲槽,则在该桶内随机选择一槽中数据按相同方式继续进行置换,直至找到空闲槽.装载率是存储数据数量与散列表槽数量的比值,用于衡量散列表当前负载情况.当装载率高时,由于剩余空闲槽数量少,插入操作中数据置换频发,导致 CHT 频繁访问内存,同时 CPU 中出现大量临时性缓存,散列表性能受到影响.

在基于键值的内存缓存系统中,这类系统主要的工作负载特点是读操作相比写操作更为频繁[10,11].当读操作的请求未命中时,系统将读操作请求的数据写入内存缓存[12].当未命中的读操作请求数量增多时,由于频繁写入缓存未命中的数据,导致了基于键值的内存缓存系统工作负载引入了更多的替换操作.当散列表装载率高时,基于键值的内存缓存系统 CHT 频发,导致了大量内存访问,降低了缓存系统整体性能.因此,基于键值的内存缓存系统散列索引方法在减少写操作访问内存的同时,需要保证系统的命中率,减少写操作数量.随着内存缓存数据量的大幅增加,在多核 CPU 上并行散列表结构的索引系统已经逐渐出现性能瓶颈,亟需进一步改进散列表的高性能和可扩展性.

硬件的发展与变迁推动着系统软件的演化[7].随着 GPU 硬件计算能力和并发性能的提升,更多的系统任务在 GPU 上运行.键值存储缓存系统和关系型数据库中的 CHT 相关工作,在 GPU 等硬件上性能得到了一定的提升[13-17].然而,在现有的 CHT 在多核 CPU 和 GPU 硬件上的并行优化方法中,系统在获取数据时的操作并不高效,计算系统的性能仍然受到内存带宽的限制[18-21].首先,由于散列表的稀疏性和随机性,在访问数据保存在芯片缓存后,在下次访问前,被其他访问的数据替换逐出芯片缓存,导致了芯片缓存的低效利用以及大量获取数据的内存访问,系统性能受到影响.Xie 等[49,50]提出了针对 GPU 的 cache bypassing 方法来减少芯片缓存的低效使用.其次,在采用 GPU 硬件作为计算核心单元时,在 CPU 与 GPU 内存之间的频繁数据传输引入了更多的系统中断,进程切换,驱动调用等额外开销.因此,减少散列表内存访问次数与减少数据在 GPU 访存与内存之间的频繁移

动仍是优化散列表系统性能的重要途径.

在上述背景下,本文提出了一种适应 GPU 的混合访问缓存索引框架 CCHT（Cache Cuckoo Hash Table）.该索引框架可应用于缓存替换系统进行索引管理.CCHT 通过多级缓存索引数据结构与支持并发读写的缓存插入查询算法实现了较低的内存访问次数.多级索引数据结构在提供了索引位置信息与全部缓存踢出优先队列信息的同时,提供了散列表桶内踢出优先队列.支持并发读写的缓存插入查询算法，给出了在内存缓存系统中插入和查询数据时散列表的操作方法.包括当散列表桶内索引存储空间满与全局存储空间满时的踢出算法和插入查询时桶内踢出队列与全局踢出队列的更新算法.本文根据内存缓存系统的命中率与索引空间占用开销,分别提出了命中率相对更高的双重 LRU CCHT 与缓存空间开销占用相对更小的粗粒度 LRU CCHT.其中,双重 LRU CCHT 通过两个缓存踢出优先级队列,在全局踢出与桶内踢出过程中独立使用,实现了高缓存命中率.粗粒度 LRU CCHT 方法则仅通过一个缓存踢出优先级队列,在全局踢出与桶内踢出过程中共同使用,在保证了缓存命中率的同时,进一步减少了优先级队列对索引空间的占用开销.本文将 CCHT 在 GPUCPU 异构环境下进行了实现,并为减少内存带宽的占用与 GPU 的访问次数,提出了基于外存计算系统(out-of-core)的多级索引数据结构.通过这种结构,实现了在 CCHT 的请求处理过程中,在 CPU 与 GPU 内存间仅传输键数据,避免了值数据占用 GPU 内存空间以及带宽资源.通过实验验证,当缓存空间大小与散列表比值为 80%时,插入与全局负载的平均访存次数分别最高降低了 30.39%和 32.91%;当缓存空间大小与散列表比值为 90%时,分别降低了 94.63%,97.29%.表明 CCHT 在散列表高装载率的情况下,仍能提供较低的内存访问次数,保证了缓存替换系统的性能.在 CPUGPU 异构环境上的实验说明,CCHT 相对其他数据索引方法,系统吞吐性能上的得到了提升,验证了采用的多级索引数据结构与实现方法的有效性.

1 相关工作

Cuckoo 散列是一种高效空间利用率的开放寻址的方法[6].它对每个对象指定多个候选散列表桶位置,同时允许将已存储的对象替换到其他候选桶位置中.SILT[22]通过两个散列方法实现了空间的高占用率,但受到了散列表大小的限制,不能应用于大存储空间的内存缓存中.MemC3[4]在保证空间的高利用率的同时,通过标签与优化锁机制,消除了对散列表尺寸大小限制.Hopscotch Hashing [23]和 Cache-Oblivious hashing[24]通过增加索引指针空间保证访问并发性,提高了吞吐性能.Li[29]等通过硬件事务内存（HTM）与粗粒度锁机制,实现了访问的高并发.FlashStore [24]通过 Cuckoo 散列将一个对象映射到 16 个桶位置上,提高插入时对象寻找空闲槽的几率.Kirsch[25]等采用了附加隐藏空间减少了插入寻找空闲空间的开销.

在 CPUGPU 异构环境上 CHT 实现研究中,Horton table[16]采用了附加空间与附加散列映射函数,代替了原有无空闲槽时进行的替换方法.Stadium Hashing[26]采用了 CPU 和 GPU 混合使用的方式提升性能,即将键（key）存储在 GPU 中,值（value）存储在 CPU 内存中.它通过添加票板（ticket board）的辅助数据结构来区分对于 CPU 和 GPU 的操作,同时允许对同一散列表的并发读写操作.Barber[33]采用了相似的位映射结构实现了两个压缩类散列表.Xie 等[49][50]采用 GPU 上程序编译和插桩的方式静态或动态指定临时或不常用的数据跳过部分芯片缓存,以减少芯片缓存的频繁置换.

散列表在应用中,广泛使用于各种类型的系统软件及数据处理程序.很多内存键值存储系统通过应用于 GPU 上的散列表加速键数据的查找[15,34,35].早期实现在 GPU 上的散列表用于数据库操作,图形处理和计算机视觉[36-40].在内存数据库中,有很多相关的工作在 CPU 平台[27,41-43]、GPU 平台[44-46]和 Xeon Phi 平台[13,47]上对散列表进行优化.

2 研究背景

在本节,介绍了论文工作的基础.其中 2.1 描述了 CHT 的基本概念.2.2 给出了 CHT 的典型实现方式.2.3 描述了基于 LRU 的缓存替换策略.2.4 给出了 CUDA 编程技术基本概念.

2.1 CHT基本概念

典型的 CHT 包含两个散列方法对键值数据（KV）进行映射[6].图 1 给出了两个典型的插入场景,即将两个不同的键值数据 KV_1 和 KV_2 插入到 CHT 中.其中,行号标明了散列索引的位置信息,每个位置的桶内按照典型的 CHT 设置有 4 个槽对键值数据进行存储.例如,H_1 和 H_2 表示两个独立的散列方法,每个键值数据可以通过这两个散列方法映射到两个不同的候选位置上.对于 KV_1,映射到位置 0 和位置 2,对于 KV_2,映射到位置 3 和位置 5.当插入 KV_1 时,通过两个散列方法找到对应的候选位置 0 和 2,这两个位置的桶中均有空闲槽用于插入新的键值数据.根据具体 CHT 的算法,选择 0 或 2 的位置的桶进行插入.

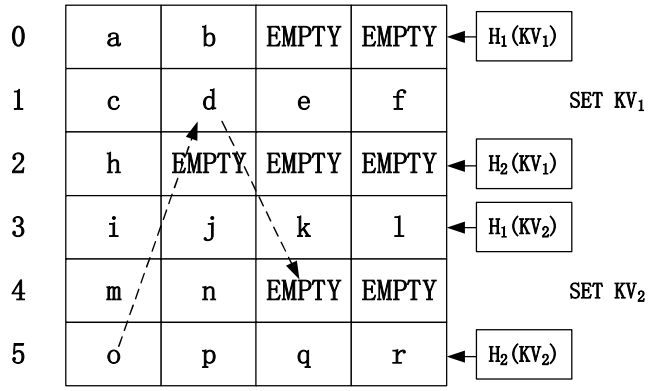

Fig. 1 Set KV_1 and KV_2 on a CHT

图 1 在 CHT 中插入 KV_1 和 KV_2

当插入 KV_2 时,通过两个散列方法找到对应位置 3 和 5,发现 3 和 5 位置的桶中已无空闲槽.当遇到此种情况时,CHT 会在候选位置的桶中选择一个槽内的键值数据进行踢出,将新的数据放入该槽内,同时通过散列方法计算踢出键值数据的另一候选位置,如果有空闲槽,则放入,如果没有,则重复踢出置换操作,直至找到空闲的槽.典型的 CHT 设有踢出置换次数阈值,当达到阈值后,停止踢出替换操作,CHT 进行扩容.CHT 踢出置换操作的引入,使 CHT 中的空闲槽能更多的被使用,从而 CHT 的负载率可以大于 95%.在 KV_2 的插入场景中,由于对应候选位置 3 和 5 的桶中已无空闲槽,则随机选择位置 5 桶中槽内值为 o 的键值数据进行踢出置换,将 KV_2 放入原有 5 位置桶中 o 对应的槽中.通过散列方法计算 o 的另一候选桶位置为 1,查询位置 1 的桶,发现无空闲槽,则继续选择一个键值数据 d 进行踢出置换,将 o 放在原有 d 的槽空间中.再对 d 进行散列值计算,查看另一候选位置 4 的桶,有空闲槽.将 d 放入空闲槽中,结束本次插入操作.Li[29]等证明了采用宽度优先,直至找到一条可以置换的路径查找策略,是一种有效的踢出置换方式.

当进行查询时,通过 H_1 与 H_2 计算散列值,查找对应位置的桶中槽内键值数据,通过遍历与比较槽内键数据来获取存储的值数据.

CHT 仅支持同一种任务类型同时发生,如并发插入操作或并发查询操作.当插入和查询操作同时发生时,插入操作可能因无空闲槽,发生置换操作.此时,若查询正在置换的目标数据,或查询的目标数据在未正确返回前,目标数据置换入另一桶中的槽内,将造成对已索引数据的查询失败.上述两种情况导致插入与查询同时进行时,散列表丧失数据完整性.CCHT 相对于 CHT 移除了无空闲槽时的置换操作,因此支持插入和查询操作并发执行,提升了散列表的并发性能.

2.2 CHT的典型实现

CHT 在实际应用中的性能,受到大量参数的影响.在 CHT 中,关键的散列函数个数与每个位置中的存储空间个数影响了 CHT 的负载率及访问的延迟.当增加 CHT 中散列函数个数,允许键值数据映射到更多的位置上时,

由于对请求数据的候选地址变多,查找空闲槽的机会变大.这种多散列函数的方式可以有效提升 CHT 的负载率.但当进行查找操作时,由于散列函数的变多,导致了需要查看更多位置的桶与槽,导致了访问延迟的增高与处理器中 cache line 中的加载次数的增多.在实际使用过程中,CHT 的散列函数设为两个能够保证槽的充分利用和高负载率的同时,有效减少查找时间,降低访问延迟.

在 CHT 中,增加每个桶内槽空间数量,可以有效提升负载率.当桶内槽数量增加时,在一次插入和读取操作过程中访问单一桶中槽的数量增多,减少了第二次或者更多次的散列操作,有效增加了 CHT 的负载率.在 CHT 实际应用中,大部分的设计采用了每个桶配有四个或八个槽空间[6].采用这种配置的原因是一个处理器中的 cache line 可以一次缓存一个或者两个位置的数据.如果每个桶中使用更多的槽,在进行键值数据比较时,将导致更多的处理器 cache line 加载,访问延迟增加.

CHT 的主要特性是能够提供快速的查询.在响应读取请求时,最大的查找空间数为 n*s,其中 n 为散列函数个数,s 为单个桶中对应的槽空间个数.对于桶中用于槽的存储区域,采用定长存储空间的分配方式,方便通过数组或者向量的方式对数据进行访问.这种方式使 CHT 更易于在不同处理器架构上进行实现,如 CPU,GPU 和 Xeon Phi.

2.3 基于LRU的缓存替换策略

在内存缓存系统中,系统将部分数据从持久化存储介质缓存到内存中,用来降低数据访问延迟.由于内存的存储空间有限,小于所有的数据存储空间需求,当内存缓存空间占满时,需要通过缓存替换策略踢出已缓存数据,获取空闲空间存放新的缓存数据.缓存替换策略保证了更有价值的缓存数据留在内存中,通常采用命中率来衡量其有效性.在系统实现时,通常增加访问吞吐与延迟作为综合的考量指标.在实际使用中,需根据不同的应用负载选择适合的替换策略.如在 web 应用的场景下,最近热门数据被频繁访问,适用最近最少使用（LRU）替换算法[8,9].

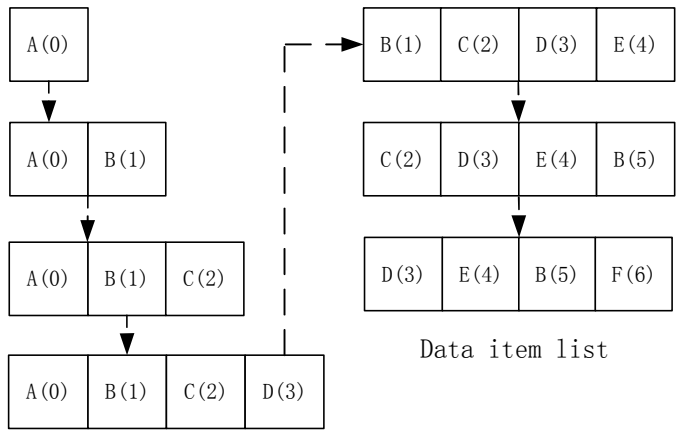

Fig. 2 Using LRU replacement police manages data items

图 2 通过 LRU 缓存替换策略管理数据单元

LRU 算法是典型的缓存替换算法.其核心思想是将最近访问的数据单元作为最有价值单元,需要替换时,踢出距离上次访问最长时间的数据单元.LRU 算法流程如图 2 所示,LRU 算法的数据单元存储空间上限为 4,存储空间右侧为最近访问的数据,同时对于每个数据单元标记操作的时间标签,操作包含插入与读取操作.在执行第 1-4 步时,在存储空间未满的状态下,依次将数据单元 A,B,C,D 插入到存储空间中.在第 5 步时,由于存储空间已满,LRU 算法选择距离上次访问最长时间的数据单元 A 进行踢出,并插入新数据单元 E;在第 6 步时,由于数据单元 B 被访问,将数据单元 B 移至存储空间最右侧,同时更新时间标签;第 7 步时,需要插入新数据单元 F,由于存储空间已满,与第 5 步相同,此步踢出最左侧数据单元 C,将 F 放入最右侧存储空间中.LRU 由于算法实现简单,在一定场景下命中率高,对应用性能影响小的特点,被缓存替换系统广泛使用,如 memcached[5], memC3[8],

MICA[9]等.

LRU 算法在实现过程中,通常采用链表和时间戳两种方式.采用链表方式进行实现时,通常与邻接散列表结合使用[5],通过数据单元内的指针快速访问散列表和 LRU 算法管理的数据结构.采用时间戳的方式,常用于固定单元大小的索引结构,通过对时间戳的比较,选取最早访问的数据单元进行踢出[31].

2.4 CUDA编程技术

计算统一设备架构（compute unified device architecture, CUDA）是显卡厂商 NVIDIA 推出的并行计算架构,已经逐步发展成为 NVIDIA GPU 的编程标准规范.

CUDA 提供了基于标准 C 语言的编程模型,支持对 GPU 操作相关的关键字与结构体.基于 CUDA 的 GPU 编程程序中包含 CPU Host 端执行代码与 GPU Device 端代码,程序中的两部分代码通过 CUDA 的编译器自动的分为两部分进行编译与链接[28].

CUDA 支持程序开发人员编写称为内核（kernel）的设备方法代码.内核被 GPU 以单指令多线程（single instruction multiple thread,SIMT）形式执行.在程序执行过程中,加载一个内核方法相当于调用内核方法,同时程序开发人员需要指定对应的 GPU 空间网格（grid）执行.一个网格包含多个线程块（block）,构成一个二维空间,每个块中包含多个线程,构成一个三维空间.每个 GPU 中的线程都通过线程标识符（thread id）进行标示.在一个块中的线程可以通过栅栏实现同步.

GPU 线程在内核方法执行过程中获得 GPU 内存的访问权限.每个线程对存储的操作包括读写私有寄存器和本地内存.内核方法中的本地变量被自动的分配到寄存器或者内存中.GPU 其他内存中的变量可以通过接口创建与管理.在 CUDA 程序中可能包含多个内核,所有的操作都可以应用于每一个内核.在下面 CCHT 设计与实现中,将所有的 CHT 与替换操作全部实现于内核中,提高程序的执行效率,同时面向单一的处理器实现能够更方便的向其他平台进行移植.

3 基于不同索引空间占用缓存索引 CCHT 方法

在本节中,我们主要介绍了基于不同索引空间占用的缓存索引方法设计与实现,包括索引结构设计与缓存替换算法.其中,3.1 节给出了面向内存缓存的散列索引分析.3.2 节描述了双重 LRU 索引访问方法.3.3 描述了粗粒度 LRU 索引访问方法.

3.1 面向内存缓存的散列索引分析

在内存缓存系统中,散列索引与缓存替换策略一般分别进行实现[5,9].散列索引用于对内存中的缓存数据进行索引,在缓存系统处理查询请求时可快速访问目标数据.常见的散列索引如本文提到的应用于 MemC3[9]中的 CHT 和 memcached[5]中的开散列方法（open hashing）.替换策略用于当内存缓存被所需缓存数据存储满时,有新的数据需要进行缓存,则在已缓存数据中通过某一策略选择待踢出数据,进行替换.常用的缓存索引方法有本文中提到的 LRU,FIFO（先进先出算法）,LFU（最近最常使用算法）,CLOCK（时钟算法）[30].其中 LRU 由于实现简单,维护方便,策略符合一般工作负载需求而被广泛使用.

当通过 CHT 进行数据索引时,每个数据对应两个可选的桶,如果两个桶中已无空闲槽,则需要在两个位置中选择随机槽内数据进行置换,之后根据置换出的键值数据进行散列值计算,可能会导致更多次的访存操作,如 2.1 节中描述.在内存缓存系统中,缓存数据无持久化存储需求.缓存替换策略可根据多维度指标,如命中率,访问延迟,空间占用率,吞吐等踢出缓存数据.因此,我们设计了内置缓存策略的 CCHT,即在内存缓存系统中,以 CHT 为基础,当索引散列表需要进行踢出置换时,调用缓存替换策略,进行踢出.我们依据索引开销与命中率需求实现了两种方法,在后文分别进行描述.

3.2 双重LRU CCHT缓存索引方法

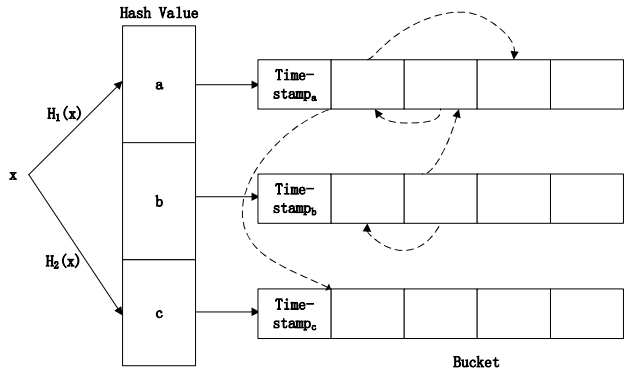

Fig. 3 Double LRU CCHT structure

图 3 双重 LRU 的 CCHT 缓存索引结构

CCHT 设计的核心思路是通过添加桶内缓存队列操作移除 CHT 原有的无空闲槽时进行的置换操作,达到减少内存访问和支持插入与查询操作并发执行的目的.

双重 LRU CCHT 缓存索引结构与典型的 LRU CHT 的结构类似.如图 3 所示,在结构中通过散列表对键值数据进行索引.每个散列值对应一个桶,每个桶中包含有固定数量的槽.每个槽中包含有键值数据,占用标示,两个用于桶内 LRU 的槽指针,两个用于全局 LRU 的槽指针.其中与典型的 LRU CHT 的结构不同的是增加用于桶内 LRU 的槽指针.为方便理解与后续方法描述,我们在图 3 中对每个桶添加时间戳标签,此时间戳等同于每个桶中 LRU 队列队尾单元的时间戳,标记每个桶中现有最早放入元素的时间.

算法 1.双重 LRU CCHT 插入算法

输入：键数据 key,值数据 value

输出：插入完成状态

```
if Existfreespace()!=true//if there exist free space
    Evictglobaltail()//evict global LRU tail unit
    UpdateLRU(H,i)
end if
H1=Hash1(key)//compute the hash value of key with hash function 1
if  i=Findfree(H1)// find a free space in the Bucket H1
    Set(H1,i,key,value)//insert key and value into free space
    UpdateLRU(H,i)// update LRU queue
    return success
end if
H2=Hash2(key)// //compute the hash value of key with hash function 2
if  i=Findfree(H2)
    Set(H2,i,key,value)
    UpdateLRU(H,i)
    return success
end if
H=Comparetimestamp(H1,H2)//find a hash value through comparing the timestamps of H1 and H2
i=EvictBucketTail(H)// evict the tail unit of bucket H LRU queue
Set(H,i,key,value)
updateLRU(H,i)
```

```
return success
Procedure UpdateLRU(H,i):
UpdateBucketLRU(H,i)//update bucket internal LRU queue
UpdateGlobalLRU(H,i)//update global LRU queue
```

算法 2.双重 LRU CCHT 查询算法

```
输入：键数据 key
输出：值数据 value
H=hash1(key)
for i=0;i<BucketInternalMax;i++ do
    if CompareKey(Bucket[H][i]->key,key)//compare the value of key
        UpdateLRU(H,i)
        return Bucket[H][i]->value
    end if
end for
H=Hash2(key)
for i=0;i<BucketInternalMax;i++ do
    if CompareKey(Bucket[H][i]->key,key)
        UpdateLRU(H,i)
        return Bucket[H][i]->value
    end if
end for
return NULL
```

在通过 CCHT 向内存缓存中插入键数据 key 和值数据 value 时,如算法 1 所示.如果内存缓存中有空闲缓存空间,通过散列函数 $Hash_1(key)$得到对应的散列值 H_1,检查 H_1 对应的桶中是否有空闲槽.如果有空闲槽,则将键值数据插入该槽,更新占用标记,并将该槽置于全局 LRU 队列队首和桶内 LRU 队列队首;如果没有空闲槽,则通过 $Hash_2(key)$计算得到散列值 H_2,检查对应桶中是否有空闲槽,如果有空闲槽,则进行上述相同操作.如果 H_1 和 H_2 对应的桶中都没有空闲槽,则比较 timestamp,选择时间戳较小的桶.如果 H_2 的时间戳小,说明散列值 H_2 对应的桶中 LRU 队尾的槽相对散列值 H_1 对应的桶中 LRU 队尾的槽更久没有被访问.踢出散列值 H_2 对应的桶中 LRU 队尾的槽,包括释放对应的键值数据,在全局 LRU 队列与桶内 LRU 队列中进行踢出.将待插入的键值数据插入该槽中,并将该槽置于全局 LRU 队列和桶内 LRU 队列队首.如果插入键值数据时,内存缓存中无空闲缓存空间,则需要先通过全局 LRU 队列踢出队尾槽对应的键值数据与 LRU 队列槽指针.

通过 CCHT 查询方法与 2.1 节中描述的通过 CHT 进行查询类似,如算法 2 所示.在返回查询结果的同时,如CCHT 中包含所需查询的键值数据,则将键值数据对应的槽放置在全局LRU队列队首和桶内LRU队列队首.

通过双重 LRU CCHT,将原有键值数据插入过程中的踢出置换操作变为缓存替换,去除了由于置换操作引发的内存访问次数和查询空闲槽的时间.双重 LRU CCHT 通过全局 LRU 队列和桶内 LRU 队列独立使用保证了内存缓存系统的命中率.这种方法相对于 LRU CHT,增加了用于桶内 LRU 队列的指针存储空间开销.因此我们提出了粗粒度 LRU CCHT,相对双重 LRU CCHT,节省了指针占用存储空间的开销.

3.3 粗粒度LRU CCHT方法

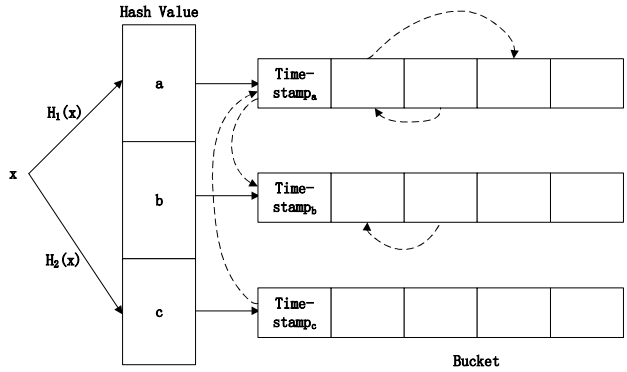

Fig. 4 Coarse LRU CCHT structure

图 4 粗粒度 LRU CCHT 缓存索引结构

为了节省 CCHT 中指针占用的存储空间,我们提出了粗粒度 LRU CCHT 方法,如图 4 所示.相对双重 LRU CCHT 索引结构,我们取消了用于全局 LRU 队列的槽指针,添加了基于桶的 LRU 队列操作.每个桶中包含有两个桶指针,用于维护粗粒度的 LRU 队列.

通过粗粒度 LRU CCHT 方法进行插入操作和查询成功时,将对应桶放置于桶 LRU 队首,对应的索引单元放置于桶内 LRU 队首.当插入数据时,若内存缓存中无空闲缓存空间,则选择桶 LRU 队列中位于队尾的桶,选择队尾桶中的桶内 LRU 队列队尾的槽进行键值数据的踢出与释放.

粗粒度 LRU CCHT 方法通过桶 LRU 算法取代了双重 LRU CCHT 方法与 LRU CHT 方法中的全局 LRU 索引.相对于双重 LRU CCHT 减少了 2*m*（s-1）个槽指针占用,其中 m 为 CCHT 内桶的个数,s 为单个桶中槽的个数,为 CCHT 节省了索引存储开销.

4 CCHT 在面向 CPUGPU 异构环境上的实现

根据章节 3 中 CCHT 的方法描述,在本节中,我们主要介绍了 CCHT 在 CPUGPU 异构环境上进行的实现.其中 4.1 节描述了面向 CPUGPU 异构存储下的多级数据索引结构,4.2 节给出了异构环境下 CCHT 的应用接口函数,4.3 节描述了 CCHT 的实现细节,4.4 节给出了 CCHT 在实现过程中的优化.

4.1 异构存储下的多级索引数据结构

在现实应用中,如图数据、日志数据及视频数据为值数据的散列表值数据存储空间开销大.以 GPU 内存为散列表的处理设计与性能受到了系统 PCI 总线带宽,GPU 内存大小的限制.因此,我们采用了基于外存计算系统(out-of-core)的方式设计了多级索引数据结构.在存储空间初始化时,在 CPU 内存中分配连续的存储空间.在对 GPU 上散列表进行操作时,用 CPU 上原始值数据对应的索引序列值进行表示,减少 GPU 内存空间的占用及 PCI-E 带宽传输开销.

键值数据在 CPU 和 GPU 的分布如图 5 所示.如插入键值数据对 KV_1,值数据 V_1 存储在 CPU 内存的连续值存储区,位置标识 ID 为 0;键数据 K_1 与索引位置 ID 值 0,通过系统总线传输至在 GPU 内存中,并存储在对应散列值桶中的空闲槽内.当查询 KV_3,将 K_3 通过系统总线传输至 GPU 内存中,待查询完成后,通过总线返回对应位置标识 ID 值 2.通过多级 CPUGPU 异构值索引结构,有效的减少了 GPU 中内存占用和总线带宽占用.在 GPU 内存中,除查询必须的键数据,仅包含值数据的位置标识,相对原始值数据,减少了有限 GPU 空间内存的占用.在请求处理时,插入与查询的值数据均不通过总线传输至 GPU 内存,有效的避免了因值数据过大造成的总线带宽占用以及性能损耗.

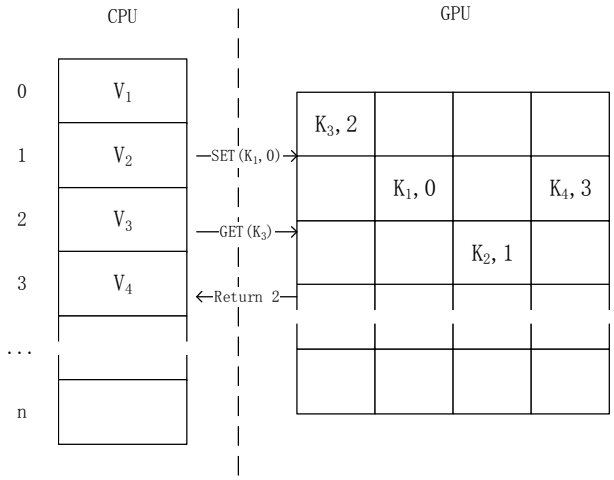

Fig.5 The CCHT data structure under CPUGPU

图 5 CPUGPU 下的 CCHT 数据结构

4.2 异构环境下CCHT的应用接口函数

CCHT 实现的方法主要包含接入库函数,数据存储区域初始化函数及操作散列表的 GPU kernel（核）函数. 我们采用的 CUDA 9.1 版本的 GPU 编程框架,代码行数共计 1385 行,其中接入库函数与数据存储区域初始化函数为 324 行,GPU kernel 函数为 953 行,其余为宏定义及全局变量索引.

Table 1 Interface and implemented methods on CPU

表 1 CPU 接口与实现函数

方法名	功能描述
set	插入键值数据
get	查询键值数据
del	删除键值数据
CCHT_init	初始化 CPU 数据结构与 GPU 存储索引区域
flush_ops	向 GPU 提交在缓冲区中待执行的操作与键值数据,其中每个任务包含请求操作类型标识

Table 2 Kernel method on GPU

表 2 GPU 核函数

方法名	功能描述
CCHT_process	global 函数,用于 CPU 向 GPU 提交键值处理任务,解析任务类型并执行相应 device 函数
CCHT_set	device 函数,用于将数据插入散列表
CCHT_get	device 函数,用于查询并获取散列表中的目标键值数据
CCHT_del	device 函数,用于删除散列表中的目标数据
CCHT_evict	device 函数,用于踢出散列表缓存队列中末尾的数据
hash1,hash2	device 函数,用于计算键数据散列值

实现的方法与功能描述如列表 1 与列表 2 所述.列表 1 给出了在 CPU 上实现的函数,主要包含用于其他 CPU 程序调用进行键值操作的接入库函数,CPU 与 GPU 存储区域索引与内容初始化的数据初始化函数,及向 GPU kernel 函数提交任务与键值数据的函数.CPU 上的函数主要其他程序提供了键值数据操作的接口,对 GPU 上的 kernel 函数进行了封装,实现了其他程序调用时对 GPU 操作的透明化.列表 2 描述了 GPU 上的 kernel 函数.主要包含了 GPU 用于接收与解析 CPU 提交键值操作任务和数据的 global 函数,散列表操作的 device 函数,缓存队列踢出操作的 device 函数.GPU 上的实现的 kernel 函数用于在 GPU 存储区域中的散列表上处理对应的键值数据.

CCHT 在 GPU 上的操作仅包括 CCHT_set、CCHT_get、CCHT_del 和 CCHT_evict 等 4 个分支核函数操作,符合 GPU 以单指令多线程(single instruction multiple thread,SIMT)形式执行.CCHT 在 CPUGPU 异构环境下执行可以获得更好的并发性能.

4.3 实现细节

在数据存储区域初始化时,采用 malloc 与 cudaMalloc 分配所需存储区域.其中包含了在 CPU 上执行缓存操作与键值数据的存储区域,GPU 执行结果返回的存储区域;在 GPU 上接受执行操作与键值数据的存储区域,以槽为单元的散列表键值数据存储区域,桶中标识的存储区域及散列表操作执行结果的存储区域.在完成分配后,通过 memset 与 cudaMemset 实现数据存储区域的初始化.

在 CPU 上的操作键值数据的函数,将其他程序调用传递进的键值数据与操作,复制至 CPU 内存中的键值数据与任务的缓冲队列中.所有操作键值数据的函数共用同一数据与任务缓冲队列,允许任务缓冲队列中包含多个写操作和多个读操作.当达到向 GPU 提交任务数的阈值时,则调用 flush_ops 函数,通过 global 函数 CCHT_process 向 GPU 传递数据及相应的操作标识.当向 GPU 提交的任务执行完成后,将结果返回至指定的存储区域.在 CPU 上运行的程序对结果进一步处理,如判断读操作是否命中,更新可用缓存空间大小等.

在 GPU 上执行核函数时,每个核函数线程执行单一任务及键值数据操作,根据 GPU 线程的全局 id 进行指定,CCHT_process 接受传递进入 GPU 内存的键值数据及任务数据,并根据任务数据的标识进行解析,以获取操作类型.对不同的散列表操作选择调用 CCHT_set、CCHT_get 或 CCHT_del,完成后将结果返回至指定的 GPU 内存存储区域.当内存缓存空间满时,CPU 传递的写操作的任务数据包含替换操作与写操作,GPU 在调用 CCHT_evict 执行踢出操作后,再调用 CCHT_set 进行写操作.

4.4 优化细节

CCHT 采用任务批处理提交方式.在其他程序调用接入库函数时,将传入的键值数据及对应的操作复制至键值数据与操作的缓冲区,当达到批处理提交的任务数阈值时,将键值数据及操作提交至 GPU 核函数,由 GPU 线程根据线程 id 并发对相应操作的键值数据进行处理.通过任务批处理提交的方式,减少了 CPU 与 GPU 间内存的访问与传输频次,减少了 PCI-E 总线的访问,同时能充分利用 GPU 多线程的并发性,提升散列表任务的处理性能.

通过 GPU 执行维度参数配置实现单一 warp 中执行一种指令. 在 CCHT 中,指令分支出现于各线程运行 global 函数 CCHT_process 后,需要根据散列表操作类型,调用不同的 device 函数.如果采用同一 warp 中混合多种操作,将导致由 warp divergence 引发的同一 warp 中的不同散列表操作类型将顺序执行.我们通过限定同一 warp 执行单一线程实现避免指令分支造成的同步等待.虽然同一 warp 中仅执行单一线程影响了 GPU 的并发性能,但基于 warp 粒度的并发仍能达到很好的吞吐效果,在实验章节 5.5 中的得到了验证.

在键值数据与任务的缓存区的实现上,采用了混合与复用的方式.所有操作的键值数据与任务共享一个连续的缓冲区.在返回结果时,将数据返回至提交任务的数据缓冲中.这种混合与复用的实现方式减少了 CPU 与 GPU 内存的空间开销,释放了更多内存空间用于键值数据的存储.同时相对多段缓存区的设计,连续的内存访问操作比例增大,减少了因为数据存储在多段缓存区域造成的内存访问.

基于 CUDA 原生的原子操作锁机制.在 CCHT 中,包含并发写入、读取槽数据与变更 LRU 队列的操作,需

要对每个槽和 LRU 队列进行读写锁设计.即需要支持多个线程同时操作数据可以获得更好的并发性能,防止因锁等待导致单个线程任务延迟过长.我们在 GPU 内存中设置了锁标识数据,默认值为 0.对于写操作,采用了原子 atomicMax()方法实现,返回现有标识数据,并将标识数据更新为现有标识数据与置入数据的最大值.当置入数据值 1.当返回值为 0 时,表明锁空闲,完成后通过 atomicExch()置 0 解锁.当返回值不为 0 时,表明有其他读或写线程正在访问数据.对于读操作,基于 atomicAdd()方法进行实现,返回现有标识数据,并将标识数据更新为现有标识数据与置入数据的和.当返回值模 2 值不为 0 时,表明有写线程正在访问数据;当返回值模 2 结果为 0 时,表明无其他写线程访问数据.成功占用锁并完成操作后,通过 atomicSub()方法释放当前读线程对锁占用.通过上述方法实现了允许同一时间单一写操作或多个读操作同时进行.基于 CUDA 原生的原子操作锁机制代替了 CUDA 传统原子数据操作赋值方法,打破了对依赖 CUDA 库函数的原子操作数据大小的上限限制. 在 CCHT 的 GPU 端处理任务请求过程中,线程同一时刻仅可能获取单个槽的锁,不会因为多个锁资源同时抢占而造成死锁.

5 实验结果与分析

实验在 CPU+GPU 异构服务器上进行,其中 CPU 为 Intel(R) Core(TM) i7-6700K,四核 4.00Ghz 主频,内存 DDR4 32GB,GPU 为 NVIDIA GeForce GTX 1080Ti,显存 11GB.

实验中采用 YCSB[48]生成的数据集进行测试,数据键值的长度为 24B,值类型为 100B,插入数据集的规模为 $3x10^6$ 个元素.数据操作请求包括三种典型的工作负载,Zipf,latest,uniform.其中 Zipf 工作负载分布偏度为 0.99,latest 工作负载为最近使用的数据请求,uniform 工作负载查询的数据概率相同.每个工作负载请求中包含两类插入与查询操作比值,分别为全部为查询操作,和查询操作占比 50%.根据内存缓存典型运行环境特征[12],在查询返回丢失后,进行插入操作.在实验过程中,先进行数据集加载与缓存预热操作,再进行工作负载请求操作.

为了对 CCHT 应用性能进行验证,我们实现了以 CCHT 为核心的缓存替换系统原型.为了对比,我们还实现了另外三种缓存索引算法：LRU CHT,LRU 开散列表（LRU open hash）,随机 CHT（random CHT）,其中 LRU CHT 与 随机 CHT 在 CPUGPU 异构平台下进行对比验证.LRU 开散列表由于动态分配数据空间的特性不适用于 CPUGPU 异构平台的初始化固定存储空间,我们在 CPU 平台进行了实现与对比验证.我们设置 CCHT 与 CHT 的散列值长度为 2^4,每个散列值对应的桶包含 4 个索引存储单元.CHT 的踢出替换操作上限为 5000 次.CCHT 批处理的阈值设为 2000.散列表装载率由缓存空间大小与散列表最大索引数量的比值表示.

5.1 插入平均访存

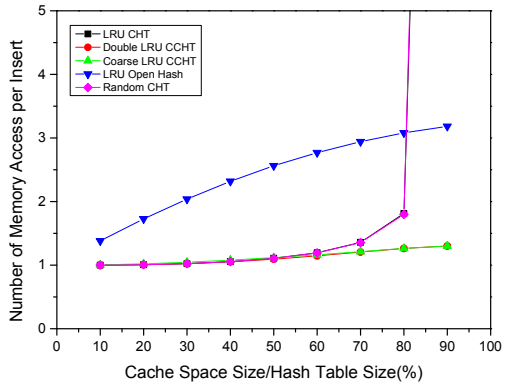

Fig. 6 Average memory access per insert with Hash table size.

图 6 插入平均访存次数随缓存空间大小变化

插入平均访存指进行数据集加载与缓存预热操作过程中,平均访问内存次数.本文中 latest,zipf,uniform 工

作负载预热与加载的数据内容与顺序相同,实验结果如图 6 所示.当缓存空间大小与散列表比值大于 60%时,双重 LRU CCHT 与粗粒度 LRU CCHT 均显著低于其他算法.当缓存空间大小与散列表大小比值为 80%时,双重 LRU CCHT 与粗粒度 LRU CCHT 的平均访存次数对比 LRU CHT 均降低了 30.39%.当缓存空间大小与散列表大小比值为 90%时,双重 LRU CCHT 与粗粒度 LRU CCHT 的平均访存次数对比 LRU CHT 均降低了 94.63%.说明 CCHT 的两种方法能有效减小内存访问次数,当缓存空间大小与散列表大小比值高时, CCHT 对比 CHT 去除了踢出与替换操作,大幅度的减少了散列表访问次数,从而减少了内存访问次数.

5.2 全局平均访存

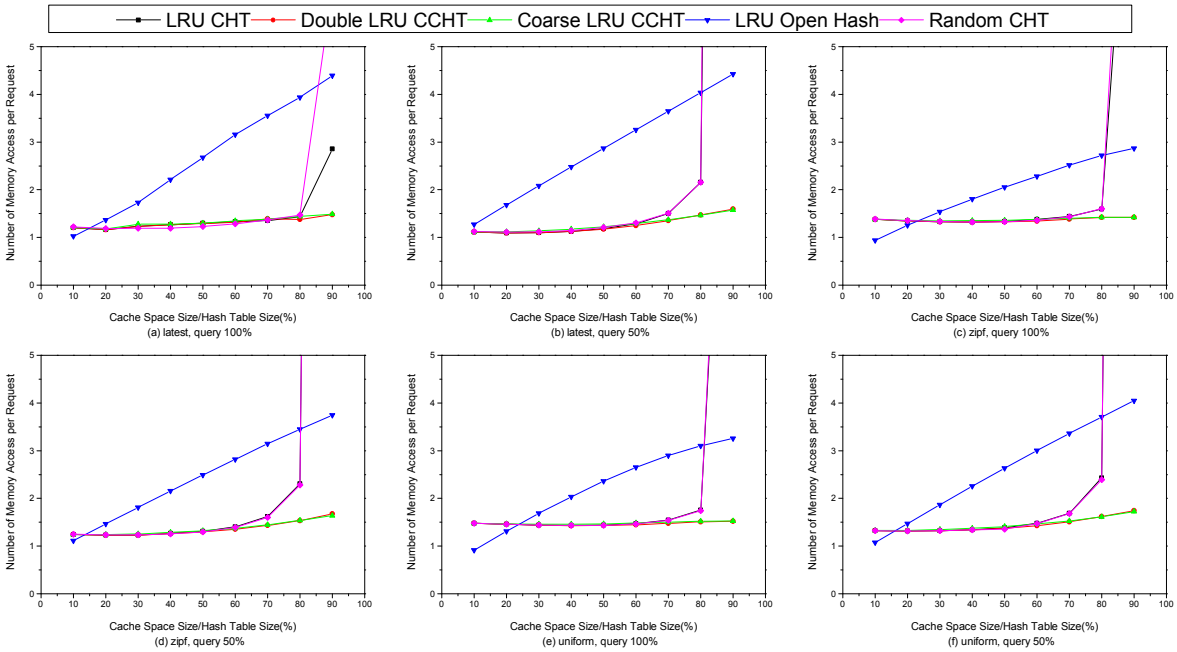

Fig.7 Average memory access per request with Hash table size

图 7 全局平均访存次数随缓存空间大小变化

全局平均访存指在工作负载测试集加载过程中,平均访问内存次数.图 7 描述了在 6 种不同的工作负载下,5 种缓存索引算法的全局平均访问内存次数随缓存空间大小变化的实验结果.当缓存空间大小与散列表比值大于 60%时,双重 LRU CCHT 与粗粒度 LRU CCHT 均低于其他算法.其中,当 query 占比为 50%时,双重 LRU CCHT 与粗粒度 LRU CCHT 效果对比 LRU CHT 更加明显的减少了平均访问内存次数.在 latest,query 50%、zipf, query 50%、uniform,query 50%中的 70%,80%,90%,双重 LRU CCHT 平均降低了 10.59%,32.86%,97.25%,粗粒度 LRU CCHT 平均降低了 9.50%,32.91%,97.29%.原因是在 query 占比为 50%的工作负载中,有大量的插入操作.随着缓存空间的增加,LRU CHT 和 随机 CHT 方法需要更多的踢出替换操作以获得空闲的索引空间,导致了大量的内存访问.而双重 LRU CCHT 和粗粒度 LRU CCHT 随着缓存空间的增加,没有访问更多的索引位置.

5.3 命中率

命中率指在工作负载测试集加载过程中,查询成功的次数占所有查询次数的比值.图 8 描述了在 6 种不同的工作负载下,5 种缓存索引算法的命中率.在 zipf 和 uniform 分布的 4 种工作负载中,双重 LRU CCHT 和粗粒度 LRU CCHT 与 LRU CHT 的命中率表现相同,命中率均随着缓存空间增加而增加.其中,双重 LRU CCHT 与 LRU CHT 的命中率差值最大不超过 0.12%,粗粒度 LRU CCHT 与 LRU CHT 的命中率差值最大不超过 0.22%.在 latest 分布的 2 种工作负载中,双重 LRU CCHT 与 LRU CHT 的命中率差值最大不超过 0.18%,粗粒度 LRU CCHT 与 LRU CHT 的命中率差值最大不超过 0.56%.CCHT 中引入了在散列表桶内无空闲槽时,通过桶内 LRU

算法踢出槽用于插入操作.CCHT 相对 LRU CHT 和 LRU 开散列表增加了缓存踢出次数,导致了命中率的偏差.同时由于采用了基于 LRU 队列的踢出策略,保证了 CCHT 的命中率相对 CHT 偏差小.

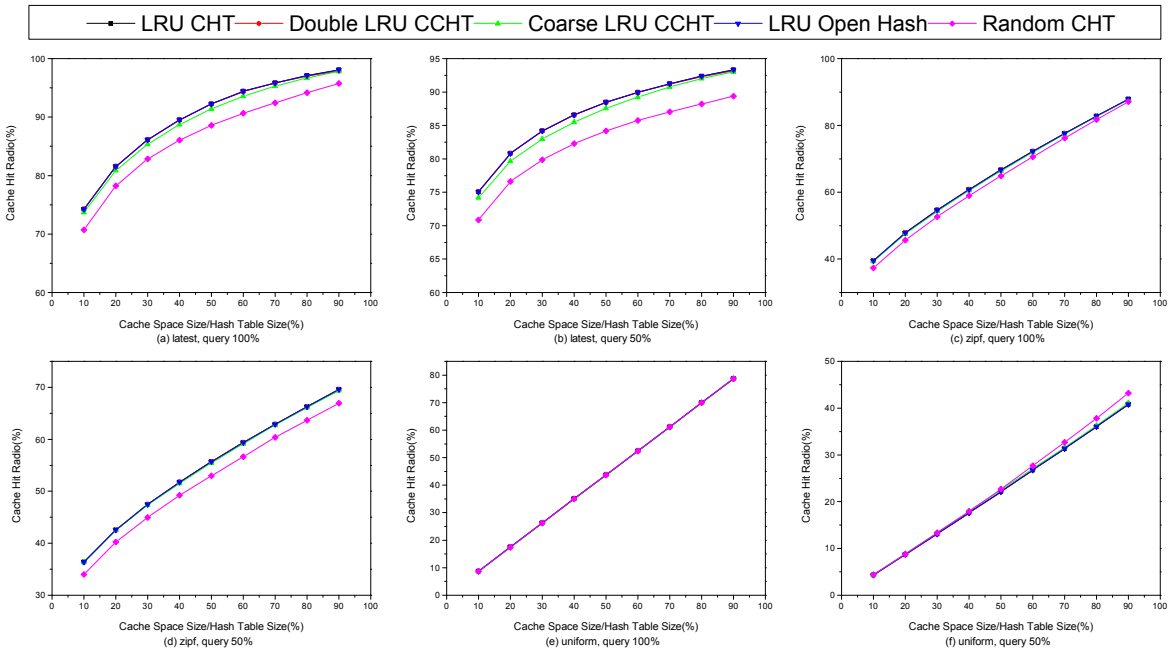

Fig.8 Cache Hit Radio with Hash table size

图 8 命中率随缓存空间大小变化

5.4 索引开销分析

Table 3 The Point Count with Different Cache Size for CCHT

表 3 CCHT 在不同缓存大小情况下的指针索引开销

Cache Space Size/Hash Table Size(%)	Double LRU CCHT Pointer Count(K)	Coarse LRU CCHT Pointer Count(K)
10	235.92	314.57
20	340.78	367.00
30	445.63	419.42
40	550.49	471.85
50	655.35	524.28
60	760.21	576.71
70	865.06	629.14
80	969.91	681.56
90	1074.77	733.99

索引开销指在存储过程中,当缓存空间满时,LRU 队列中槽索引指针的数目. CCHT 在移除 CHT 置换操作的同时,引入了用于 LRU 队列的槽指针索引空间开销.本节实验分析了双重 LRU CCHT 与 粗粒度 LRU CCHT 的在不同缓存空间大小下的索引开销,如表 3 所示.当缓存空间大小与散列表大小比值大于等于 30%时,粗粒度 LRU CCHT 的开销明显小于双重 LRU CCHT 的开销.当缓存空间大小与散列表大小的比值为 90%时,粗粒度 LRU CCHT 比双重 LRU CCHT 减少了 31.71%.当比值小于等于 20%时,粗粒度 LRU CCHT 索引开销大于双重 LRU CCHT 开销,原因是粗粒度 LRU 的一部分索引开销来源于散列表散列值对应桶之间的索引,没有随缓存空

间大小的变化而变化.当缓存空间增大后,使用的槽增多,索引开销主要来源于槽间的链接索引.而双重 LRU CCHT 的全局 LRU 队列的槽索引开销大于粗粒度 LRU CCHT 的开销,导致了两种 CCHT 算法的开销比值逐渐增加.

5.5 CCHT 在CPUGPU上的吞吐性能

我们将 CCHT 在 CPU+GPU 异构服务器环境上的吞吐性能进行了实验.在 CPUGPU 异构环境上的吞吐性能指 CCHT 每秒钟处理的请求数.在 80%缓存空间占比的工作负载下,性能实验结果如图 9 所示.在不同的工作负载中,CCHT 相对 LRU CHT、Random CHT 和 LRU 开散列表均有显著的性能提升,最高达 126.43 倍、143.17 倍和 1.78 倍.其中造成 LRU CHT 和 Random CHT 性能最低的原因是对于 CHT 的读写请求只能顺序执行,无法并发执行,进而导致每次操作类型切换时,需要通过 PCI-E 总线进行数据传输,带来了频繁的内核上下文切换与驱动调用,最终导致了性能的大福降低,甚至远低于 CPU 平台上的 LRU 开散列算法的性能.相对于 LRU CHT,CCHT 支持了所有操作类型的并发执行,最大程度的利用了 GPU 上的流处理器等硬件并发资源,同时有效减少了 CPU 与 GPU 之间由于频繁的数据传输带来的额外开销;去除了写操作在高负载率下频发的置换操作,减少了 GPU 处理过程中的内存访问次数,使散列表索引访问性能得到提升.

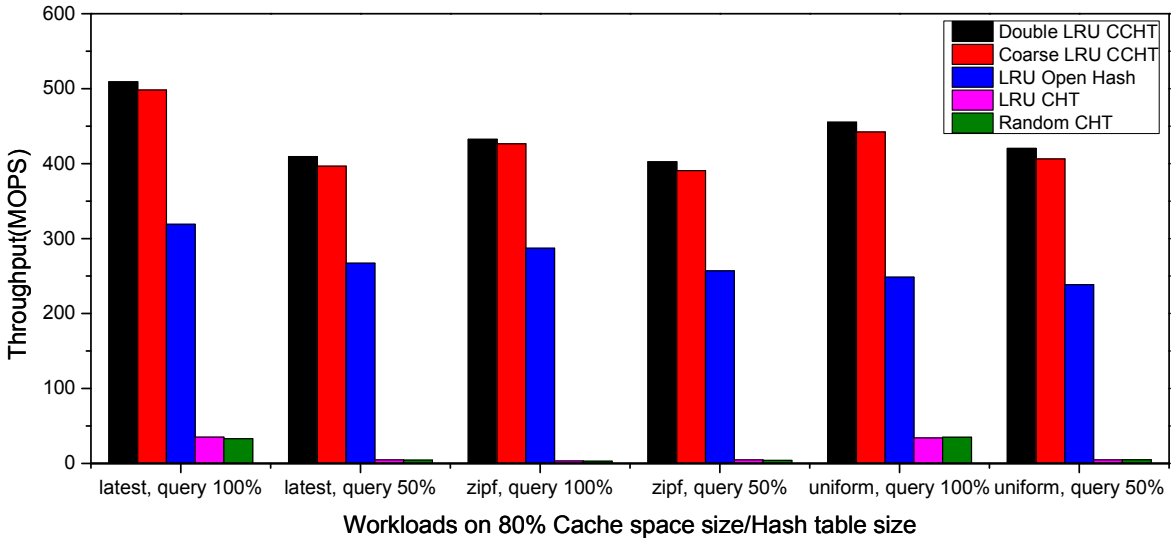

Fig.9 Performance on workloads with 80% Cache space size/ Hash table size

图 9 在 80%缓存空间工作负载中的性能表现

5.6 CCHT 在CPUGPU上的延迟

我们将 CCHT 在 CPU+GPU 异构服务器环境上的延迟进行了实验.延迟实验包括在 CPU+GPU 异构服务器环境上的数据传输延迟和单次操作的平均延迟.其中数据传输延迟表示散列表在运行过程中单次调用 GPU 时，数据在 CPU 和 GPU 内存间传输的延迟时间.在 CCHT 中具体指调用 GPU 处理散列表请求时，请求数据与操作命令缓冲区由 CPU 内存拷贝到 GPU 内存和请求结果由 GPU 内存拷贝到 CPU 内存的时间总和.数据传输延迟时间如表 4 所示，4 种散列表在各工作负载中单次调用 GPU 数据传输的规模相同，我们采用以 80%缓存空间占比的 latest，query 100%工作负载为代表进行实验验证.其中双重 LRU CCHT 和粗粒度 LRU CCHT 的数据传输延迟均高于 LRU CHT 和随机 CHT，原因是 CCHT 采用了批处理操作，允许散列表在 GPU 上大规模并发处理请求.以实验设置的批处理阈值 2000 为例，CCHT 在调用 GPU 时，单次的传输数据包括 2000 个操作数据和操作请求类型，平均单个操作数据和操作请求类型的传输延迟远低于 LRU CHT 和随机 CHT.

Table 4 CPU GPU Data Transfer Delay

表 4 CPUGPU 数据传输延时

Hash Table Type	Delay(μs)
Double LRU CCHT	40
Coarse LRU CCHT	40
LRU CHT	13
Random CHT	12

进一步，我们对单次操作的平均延迟进行了对比评估，即散列表处理单次请求在 CPU 或 GPU 环境上的平均延迟时间.其中 LRU 开散列表为单次请求处理在 CPU 环境上的平均延迟时间，其余散列表为单次请求处理在 GPU 环境上的平均延迟时间.单次操作的平均延迟如图 10 所示.双重 LRU CCHT 和粗粒度 LRU CCHT 的延迟时间高于其他散列表，原因是 CCHT 实现了 GPU 上的大规模的并发操作，虽然通过 warp 分配避免了分支等待的延迟开销，但各线程间仍存在着对于槽和 LRU 队列锁的竞争操作，以及在同一次批处理中的同步开销，即执行完操作的线程需等待未执行完操作的线程完成后一同返回.这类操作均带来了额外的开销，导致了延迟的增加.同时，从实验结果可以看出，粗粒度 LRU CCHT 延迟同样高于双重 LRU CCHT.这是因为粗粒度 LRU CCHT 在全局和桶内共用的同一 LRU 队列均在 GPU 环境上进行操作，相对双重 LRU CCHT 仅桶内锁在 GPU 环境上进行操作，粗粒度 LRU CCHT 存在更多的锁竞争. LRU CHT 和随机 CHT 虽然低于前两者，但由于受高负载下可能发生插入操作出现频繁槽数据置换的影响,单次延迟最高达到了 143μs,远高于平均延迟时间.LRU 开散列表由于没有锁竞争及运行在 CPU 环境上，延迟时间在 0.36μs 至 0.49μs.

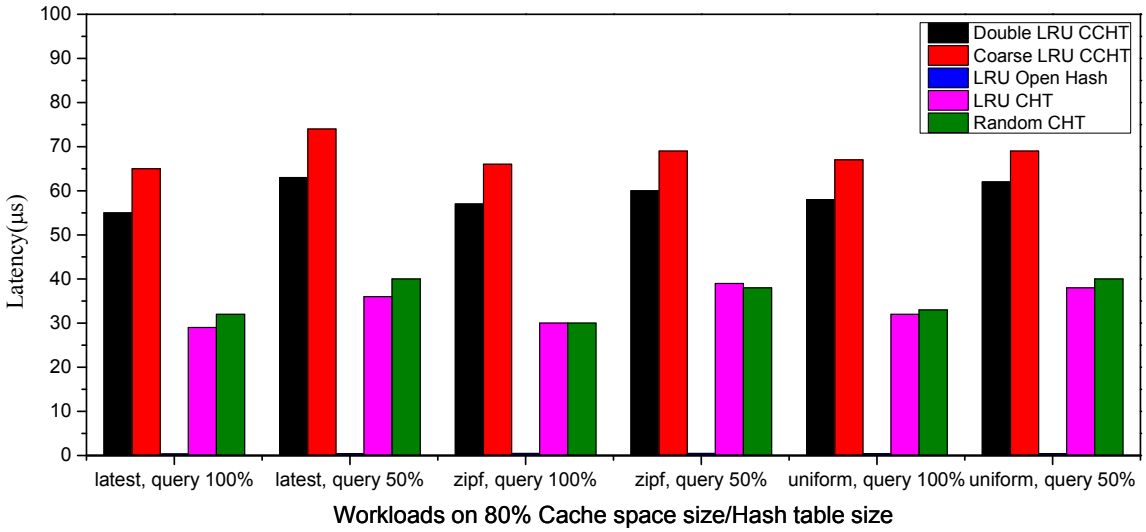

Fig.10 Latency on workloads with 80% Cache space size/ Hash table size

图 10 在 80%缓存空间工作负载中的延迟表现

6 结束语

数据索引的性能依赖于平均访问内存数.传统的 CHT 数据索引在缓存中应用时,当缓存空间大小与 CHT 空间比值高,CHT 频繁的踢出替换方法增加了内存访问数,进而对缓存性能造成了影响.本文分析了基于键值的内存缓存系统命中率与索引开销因素,依据索引指针开销给出了双重 LRU CCHT 和粗粒度 LRU CCHT,提供了用于 CPUGPU 环境下减少总线传输与 GPU 内存占用的多级索引数据结构,并在 CPUGPU 异构环境下进行了实现.通过实验验证,CCHT 在保证缓存命中率的同时,能有效的减少内存访问次数,在 GPU 上有良好的可扩展性与吞吐性能.

未来的工作我们希望进一步优化索引指针的开销,提升缓存的命中率.在 GPU 的实现方法中,考虑通过优化槽单元锁和 LRU 队列锁提升并发插入的性能,在散列任务提交任务队列前识别任务类型,预先分配任务执行所在的 warp,达到同一 warp 中并发执行多个散列表操作,达到线程粒度的并发性能,提升 GPU 硬件资源利用率.

References:

[1] Boncz PA, Manegold S, Kersten ML. Database architecture optimized for the new bottleneck: Memory access. VLDB, 1999, 99: 54-65.

[2] DeWitt DJ, Katz RH, Olken F, et al. Implementation techniques for main memory database systems. Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 1984: 1-8.

[3] Kemper A, Neumann T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. 2011 IEEE 27th International Conference on Data Engineering, 2011: 195-206.

[4] Fan B, Andersen DG, Kaminsky M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation, 2013: 371-384.

[5] Fitzpatrick B. Distributed caching with memcached. Linux Journal, 2004, 2004(124): 5.

[6] Pagh R, Rodler FF. Cuckoo hashing. European Symposium on Algorithms. Springer, Berlin, Heidelberg, 2001: 121-133.

[7] Wang HM, Mao XG, Ding B, et al. New insights into system software. Ruan Jian Xue Bao/Journal of Software, 2019,30(1):22-32 (in Chinese).

[8] Ousterhout J, Gopalan A, Gupta A, et al. The RAMCloud storage system. ACM Transactions on Computer Systems (TOCS), 2015, 33(3): 1-55.

[9] Fan B, Andersen DG, Kaminsky M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation, 2013: 371-384.

[10] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload analysis of a large-scale key-value store. Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, 2012: 53-64.

[11] Raman V, Attaluri G, Barber R, et al. DB2 with BLU acceleration: So much more than just a column store. Proceedings of the VLDB Endowment, 2013, 6(11): 1080-1091.

[12] Nishtala R, Fugal H, Grimm S, et al. Scaling memcache at facebook. Presented as Part of the 10th USENIX Symposium on Networked Systems Design and Implementation, 2013: 385-398.

[13] Polychroniou O, Raghavan A, Ross KA. Rethinking SIMD vectorization for in-memory databases. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015: 1493-1508.

[14] Ross KA. Efficient hash probes on modern processors. 2007 IEEE 23rd International Conference on Data Engineering. IEEE, 2007: 1297-1301.

[15] Zhang K, Wang K, Yuan Y, et al. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. Proceedings of the VLDB Endowment, 2015, 8(11): 1226-1237.

[16] Breslow AD, Zhang DP, Greathouse JL, et al. Horton tables: Fast hash tables for in-memory data-intensive computing. 2016 USENIX Annual Technical Conference, 2016: 281-294.

[17] Sun Y, Hua Y, Jiang S, et al. SmartCuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. 2017 USENIX Annual Technical Conference, 2017: 553-565.

[18] Ailamaki A, DeWitt DJ, Hill MD, et al. DBMSs on a modern processor: Where does time go?. VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh,

Scotland, UK, 1999: 266-277.

[19] Boncz PA, Kersten ML, Manegold S. Breaking the memory wall in MonetDB. Communications of the ACM, 2008, 51(12): 77-85.

[20] McKee SA. Reflections on the memory wall. Proceedings of the 1st Conference on Computing Frontiers, 2004: 162.

[21] Wulf WA, McKee SA. Hitting the memory wall: Implications of the obvious. ACM SIGARCH Computer Architecture News, 1995, 23(1): 20-24.

[22] Lim H, Fan B, Andersen DG, et al. SILT: A memory-efficient, high-performance key-value store. Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011: 1-13.

[23] Herlihy M, Shavit N, Tzafrir M. Hopscotch hashing. International Symposium on Distributed Computing. Springer, Berlin, Heidelberg, 2008: 350-364.

[24] Pagh R, Wei Z, Yi K, et al. Cache-oblivious hashing. Algorithmica, 2014, 69(4): 864-883.

[25] Debnath B, Sengupta S, Li J. FlashStore: High throughput persistent key-value store. Proceedings of the VLDB Endowment, 2010, 3(1-2): 1414-1425.

[26] Khorasani F, Belviranli ME, Gupta R, et al. Stadium hashing: Scalable and flexible hashing on GPUs. 2015 International Conference on Parallel Architecture and Compilation (PACT). IEEE, 2015: 63-74.

[27] Metreveli Z, Zeldovich N, Kaashoek MF. Cphash: A cache-partitioned hash table. ACM SIGPLAN Notices, 2012, 47(8): 319-320.

[28] NVIDIA Corporation. NVIDIA CUDA Programming Guide, version 10.0: Reference Description[OL] [2020-01-22] https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[29] Li X, Andersen DG, Kaminsky M, et al. Algorithmic improvements for fast concurrent cuckoo hashing. Proceedings of the Ninth European Conference on Computer Systems, 2014: 1-14.

[30] Corbato FJ. A paging experiment with the multics system. Massachusetts Inst of Tech Cambridge Project Mac, 1968.

[31] Sanchez D, Kozyrakis C. The ZCache: Decoupling ways and associativity. 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture. IEEE, 2010: 187-198.

[32] Ma YZ, Meng XF. Research on indexing for cloud data management. Ruan Jian Xue Bao/Journal of Software, 2015,26(1):145−166 (in Chinese)

[33] Barber R, Lohman G, Pandis I, et al. Memory-efficient hash joins. Proceedings of the VLDB Endowment, 2014, 8(4): 353-364.

[34] Hetherington TH, O'Connor M, Aamodt T M. Memcachedgpu: Scaling-up scale-out key-value stores. Proceedings of the Sixth ACM Symposium on Cloud Computing, 2015: 43-57.

[35] Hetherington TH, Rogers TG, Hsu L, et al. Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems. 2012 IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, 2012: 88-98.

[36] Alcantara DA, Sharf A, Abbasinejad F, et al. Real-time parallel hashing on the GPU. ACM Transactions on Graphics (TOG), 2009, 28(5): 1-9.

[37] Alcantara DA, Volkov V, Sengupta S, et al. Building an efficient hash table on the GPU. GPU Computing Gems Jade Edition. Morgan Kaufmann, 2012: 39-53.

[38] García I, Lefebvre S, Hornus S, et al. Coherent parallel hashing. ACM Transactions on Graphics (TOG), 2011, 30(6): 1-8.

[39] Korman S, Avidan S. Coherency sensitive hashing. IEEE Transactions on Pattern Analysis and Machine

Intelligence, 2015, 38(6): 1099-1112.

[40] Lefebvre S, Hoppe H. Perfect spatial hashing. ACM Transactions on Graphics (TOG), 2006, 25(3): 579-588.

[41] Balkesen C, Alonso G, Teubner J, et al. Multi-core, main-memory joins: Sort vs. hash revisited. Proceedings of the VLDB Endowment, 2013, 7(1): 85-96.

[42] Balkesen C, Teubner J, Alonso G, et al. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE, 2013: 362-373.

[43] Blanas S, Li Y, Patel JM. Design and evaluation of main memory hash join algorithms for multi-core CPUs. Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, 2011: 37-48.

[44] He B, Yang K, Fang R, et al. Relational joins on graphics processors. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, 2008: 511-524.

[45] He J, Lu M, He B. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. Proceedings of the VLDB Endowment, 2013, 6(10): 889-900.

[46] Keckler SW, Dally WJ, Khailany B, et al. GPUs and the future of parallel computing. IEEE Micro, 2011, 31(5): 7-17.

[47] Jha S, He B, Lu M, et al. Improving main memory hash joins on intel xeon phi processors: An experimental approach. Proceedings of the VLDB Endowment, 2015, 8(6): 642-653.

[48] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB.Proceedings of the 1st ACM symposium on Cloud computing. 2010: 143-154.

[49] Xie X, Liang Y, Sun G, et al. An efficient compiler framework for cache bypassing on GPU. IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2013: 516-523.

[50] Xie X, Liang Y, Wang Y, et al. Coordinated static and dynamic cache bypassing for GPUs. IEEE 21st International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2015: 76-88.

附中文参考文献:

[7] 王怀民,毛晓光,丁博,沈洁,罗磊,任怡.系统软件新洞察.软件学报,2019,30(1):22-32.

[32] 马友忠,孟小峰.云数据管理索引技术研究.软件学报,2015,26(1):145−166